Cloud Native Network Functions

Design, Architecture and Technology Landscape

November 2019

metaswitch

metaswitch

Contents

Introduction	3
A Brief History of Virtualization and Cloud	4
The Emergence of Cloud Native	5
The Cloud Native Software Ecosystem	6
The Key Features of Cloud Native Application Architecture	7
Stateless Processing	
Microservices	9
Containers	10
Design for Automation	12
Building Cloud Native Network Functions	14
Cloud Native Control Plane Functions	
Cloud Native Data Plane Functions	
Declarative Configuration of NFs	16
Deployment Environment for Cloud Native Network Functions	17
Testing Cloud Native Claims	18
No Evolution Path from VNF to CNF	
The Cloud Native Scorecard	
Conclusion	21

Introduction

There is broad consensus in the industry that Networks Functions Virtualization, as defined in the original white paper published by 13 telcos in October 2012, has largely failed to deliver on its promises of substantial Opex and Capex reductions, together with rapid acceleration of innovation in network services and operations.

There is also broad consensus about the main reason for this failure. The white paper described a vision in which physical boxes are replaced by software appliances designed to run on commodity hardware in a virtualized environment. This encouraged vendors to port the software from their proprietary systems onto commodity hardware, but to make no other fundamental changes, for example to make it easy to automate operations—so Opex savings are not easily achieved. Not being designed from scratch to run on commodity hardware, the software often performs poorly, and consumes excessive hardware resources—making it difficult to achieve useful Capex savings. And the monolithic architecture of this legacy software suffers from the same long release cycles and heavy testing burden as the physical boxes it replaced, so innovation proceeds at the same glacial pace as it always has.

Not only does the industry broadly agree on the reasons for the failure of NFV to date, it also agrees about the right way forward. This is to build network functions as software systems designed from the ground up for the cloud, in the same way as the big Web-scale players would approach the problem: the so-called cloud native approach.

Agreement on this point is so universal that every vendor of network function software now claims that its NF products are cloud native: CNFs (Cloud Native Network Functions) rather than VNFs. The great majority of these claims are highly suspect. Squeezing a monolithic stateful legacy VNF into a container does not make it a CNF. Nor does bolting on a Web Services API. The benefits of a cloud native approach to network functions derive from a number of absolutely fundamental differences in software architecture between legacy and cloud native. These differences are so great that it's not technically or economically feasible to evolve a legacy VNF to become a CNF. In other words, to build a CNF, you have to start from scratch.

But what exactly are these differences? What truly distinguishes a CNF from a VNF? These are the questions that this white paper explores in depth.

This paper focuses on the technical aspects of CNFs: design, architecture and technology landscape. Readers who would like to understand more about what it's like to work with CNFs versus VNFs from an operations standpoint, why the cloud native approach is so superior, and how to make a successful transition from VNFs to CNFs, are encouraged to read our white paper "The Cloud Native Telco."

A Brief History of Virtualization and Cloud

Virtualization has a long history in the computer industry, and first became a mainstream commercial technology in the mid-1960s on IBM mainframes.

The modern era of virtualization was ushered in by the addition of hardware support for virtualization on x86 processors by Intel in 2005-06, which paved the way for the introduction of successful hypervisor products such as VMware. Up to this point, IT shops had installed a separate physical machine for each different server application that they deployed, with the result that most machines were severely under-utilized. With a hypervisor they could safely deploy multiple server applications per host, consolidating their resources and achieving very substantial Capex and Opex savings.

At that time, it was common to see a mix of many different operating systems in use for IT applications: various flavors of Unix, Solaris, Windows and early versions of Linux. Naturally, there was a requirement to be able to mix applications with different operating systems on the same physical host. So hypervisors were designed to expose to applications an emulation of a complete physical x86 server: a virtual machine. The server application, together with the operating system it depends on, runs inside the virtual machine, safely and securely partitioned from other server applications and their supporting guest operating systems deployed on the same host.

The business case for virtualization was a compelling one, and by 2013 more than half of all IT workloads were running virtualized. IT shops began to view their physical servers not so much as a collection of individual machines, but more as a pool of computing resources. When they deploy a server application, they don't much care which particular machine it runs on, so long as it has sufficient resources to perform as required. This is led to the introduction of infrastructure software solutions that treat a collection of x86 machines as an interchangeable pool of resources, and manage the deployment of applications on it: a cloud.

Cloud technology enables compute resources to be treated as a utility, and this opens up the possibility of a market in which compute power can be bought and sold: the public cloud. Economies of scale mean that very large providers of public cloud services can offer compute power at considerably lower cost than can be achieved in small-scale private clouds. As a result, some IT shops now choose to deploy some or all of their applications on public cloud services.

For most of the first decade of cloud technology, the great majority of applications deployed in both public and private clouds were originally written to run on dedicated, bare metal servers. Cloud services offering virtual machines that emulate physical servers, so called Infrastructure as a Service, provide an ideal environment into which such applications can be moved.

The Emergence of Cloud Native

The availability of inexpensive pay-as-you-go compute power in large-scale public clouds opened up a completely new kind of opportunity for entrepreneurs: the ability to rapidly create and roll out network-based services that could be offered to the public at scale, particularly in the realms of social media, messaging, media distribution, e-commerce and the "gig economy". In particular, it massively reduced the amount of capital risk associated with starting up and scaling such services.

The new ventures that set out to take advantage of this opportunity were not writing software to run on dedicated servers, and then deploying it on virtual machines in the cloud. Instead, they viewed the cloud as an entirely new kind of distributed computing environment that opened up exciting possibilities for new application architectures.

What these cloud application developers sought, above all, was scalability. They wanted to be able to deploy systems that would scale rapidly through many orders of magnitude with as few limitations as possible, and without the requirement to re-visit fundamental aspects of application architecture along the way. They also wanted resilience and fault tolerance; they recognized that failures can occur at every level of the stack, from individual servers to entire data centers, and from individual virtual machines to entire cloud instances, and they needed to come up with software architectures that would survive multiple such failures and continue to deliver services. But they didn't want to buy fault tolerance in the traditional way by doubling up resource usage. Rather, they expected to absorb the impact of failures through modest amounts of surplus capacity combined with automated self-healing capabilities.

In addition to scalability and fault tolerance, cloud application developers wanted to be able to evolve their software solutions quickly to meet new and emerging service requirements. In practice this meant making it possible for multiple teams to work on the software simultaneously without tripping over each other, leading to the concept of decomposing complex services into loosely-coupled components that talk to each other through open, language-agnostic APIs.

These were all difficult and challenging problems to solve, but the successful pioneers in cloud-based application development employed some of the best brains in the software industry, and there was a good deal of cooperation and sharing of learnings among them. The design patterns of what came to be known as cloud native software architecture have emerged over the last few years as a consensus within this community.

The Cloud Native Software Ecosystem

Open source software has played a very important part in the emergence of the cloud native movement, from two points of view:

Building blocks for rapid prototyping and creation of cloud native applications.

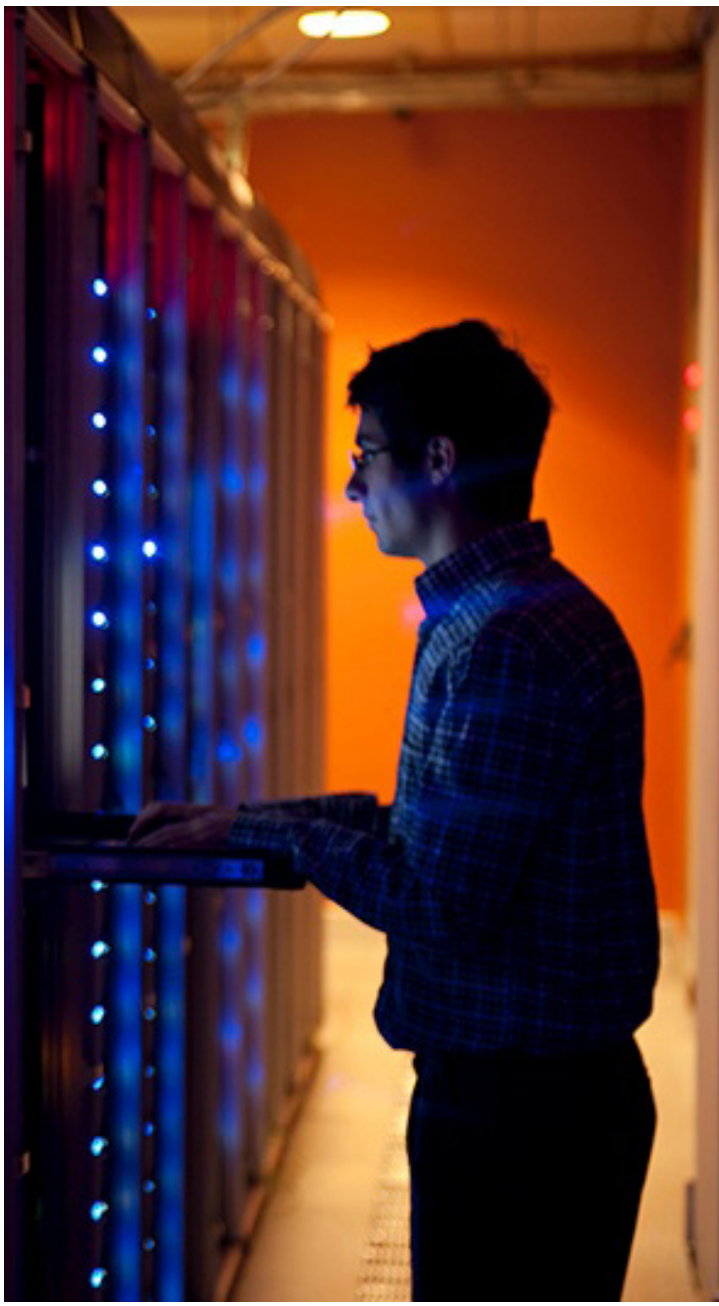
For example, storage (Cassandra, MongoDB, etcd), security (OpenSSL), APIs (gRPC, Thrift), message streaming (NATS, Kafka), service mesh (Envoy, Istio, Linkerd)

Tools for automating, orchestrating and operationalizing cloud native applications.

The primary example is Kubernetes which performs lifecycle management of cloud native applications, but other key projects include Prometheus (collection and storage of metrics), Grafana (visualization tool for metrics), Fluentd (collection and analysis of logs) and Helm (Kubernetes package management).

Many of these open source projects are hosted by the Cloud Native Computing Foundation (CNCF), a part of the Linux Foundation. The CNCF defines cloud native as follows:

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.



CNCF also publishes a very useful "trail map" that provides guidance on best practices for cloud native application development. CNFs that do not substantially embody the practices identified in this trail map have no right to call themselves cloud native.

The Key Features of Cloud Native Application Architecture

Stateless Processing

The requirement for easy scaling across many orders of magnitude is the driver behind the single most important concept in cloud native architecture: stateless processing.

The concept of stateless processing can be described as follows. A transaction processing system is divided into two tiers. One tier comprises a variable number of identical transaction processing elements that do not store any long-lasting state. The other tier comprises a scalable storage system based on a variable number of elements that store state information securely and redundantly. The transaction processing elements read relevant state information from the state store as required to process any given transaction, and if any state information is updated in the course of processing that transaction, they write the updated state back to the store.

It's probably not obvious from reading the description above how this approach enables massive scalability. So let's use a practical example to illustrate.

Suppose we are developing an e-commerce application. The application needs to support a number of HTTP transaction types including login to account, add item to shopping basket, review shopping basket, checkout etc. The application code that processes these transactions needs access to certain information (i.e. "state"), for example details of the user's account and the current contents of the shopping basket. In a traditional application architecture, this state would be kept in the application's local storage.

The first problem that we need to solve is how to provide fault tolerance. If a server dies, then any local state that is stored in it is lost. The physical servers that are deployed in cloud environments are not particularly reliable, and failures are fairly frequent. Users get pretty upset if they've spent 30 minutes online grocery shopping, and their shopping basket suddenly disappears. We face a difficult choice here: either we accept the risk that a small proportion of e-commerce sessions will fail due to equipment failure, or we have to deploy a second server to act as a backup, and maintain a shadow copy of all the state on it – which doubles the amount of hardware resources the application is consuming.

Now sppose that we need this application to support millions of concurrent online shopping sessions. A single server (or active-standby pair of servers) is not going to be able to handle the load, so we need to deploy a number of servers. The problem that we now need to solve is that each incoming HTTP request needs to be directed to the correct server, the one that knows about this particular user and session. We therefore need to deploy something like a load-balancer in front of our collection of servers, and the load-balancer needs to be able to identify the user and session from the information in each incoming request, remember which server is handling each user session, and re-direct each request to the correct server. The load-balancer is therefore quite a complex application in its own right. And because it's potentially a single point of failure, it needs to be fault tolerant, which makes it even more complex. But the biggest single issue here is that the performance and capacity of the load-balancer puts an upper limit on the transaction processing load that we can handle. What happens if our e-commerce site is wildly successful and we cannot obtain a load-balancer that is powerful enough to handle all of the demand?

With the stateless processing approach, we implement the elements that process HTTP transactions without any local state storage, and have them read and write state to and from a separate storage system. When an HTTP request arrives at one of these elements, it extracts some information from the request that uniquely identifies the session (for example, from a cookie), and then uses this information to retrieve the current state associated with this session (user account details, contents of shopping basket) from the state store. If the transaction has the effect of changing any of this state, for example because the user added an item to her shopping basket, then the transaction processing element writes the updated state back to the state store.

The difference now is that any incoming HTTP request can be handled by any arbitrary instance of the transaction processing element. We do not have to steer each request to the instance that "knows" about it, because knowledge about each session is available to every processing element instance from the state store. We still need some way to balance the load of incoming requests across the population of transaction processing elements, but we can do this without having to deploy a load-balancer, for example by leveraging DNS to perform dumb round-robin load balancing. By eliminating the load-balancer, we've eliminated the limiting factor on scale. We also don't need to worry about any individual transaction processing element failing. Such failures do not result in the loss of any state, because all the state is stored separately.

With the stateless processing approach, we implement the elements that process HTTP transactions without any local state storage, and have them read and write state to and from a separate storage system.

The stateless approach is therefore inherently fault tolerant. If any processing element instance dies or becomes unresponsive, then the built-in retry mechanisms of HTTP will result in subsequent attempts being handled by another instance. So long as we have a modest amount of performance headroom in our population of processing elements, the failure of any one of them has no impact on the service: the load that it would otherwise have handled is simply re-distributed across the remaining instances. We can very easily extend this fault tolerance mechanism across multiple data centers, so that even the loss of an entire data center will not bring down our service.

Individual processing elements can be quite small in scale: we can keep the architecture of these elements simple by not worrying about trying to make them very powerful, for example with support for lots of multi-core parallelism. We handle scaling by deploying as many processing element instances as we need to handle the load, an approach which is known as "scale out" (in contrast to "scale up", which involves deploying bigger server instances). We can also change the number of processing elements on the fly (scaling both out and in) in response to changing load – enabling us to make the most efficient use of compute resources at all times.

All of this depends, of course, on our ability to build and deploy a highly scalable and very fault-tolerant storage system in which to keep all of our application state. Because this is an absolutely fundamental requirement of the stateless processing design pattern, there has been a lot of investment in this area, particularly by the main Web-scale players. Many of the solutions that they have built to address this need are available as open source. For example, one of the leading distributed state stores, Apache Cassandra, was originally developed at Facebook, and is now used by Netflix, Twitter, Instagram and Webex among many others. Test results published by Netflix show Cassandra performance scaling linearly with number of nodes up to 300, and handling over a million writes per second with 3-way redundancy – more than enough to handle the needs of most telco-style services even with many hundreds of millions of subscribers. Cassandra includes support for efficient state replication between geographically separate locations, and therefore provides an excellent basis for extremely resilient geo-redundant services.

It's perhaps worth pointing out that stateless processing is by no means the only design pattern seen in cloud native applications, although it's definitely the most prominent. Other design patterns worth mentioning include stream processing (based on frameworks such as Heron or Storm) and serverless processing, best exemplified by Amazon Lambda. These have only emerged relatively recently, and won't be discussed further in this document – but they definitely have potential to advance the state of the art in CNF design.

Microservices

After stateless processing, the second most frequently cited aspect of the cloud native approach to software design is microservices, defined as follows:

> ***Microservices is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are highly decoupled and focus on doing a single small task well, facilitating a modular approach to system-building.***

Microservices is a big topic: entire books have been written about it, and we only have room for a brief summary here. The main benefits of a microservices approach are as follows:

Composability and reusability

Microservices encourages the development of modular software components, each of which performs a very specific task that is exposed via an open and well-documented API. Components built this way lend themselves to easy re-use in a variety of different circumstances, enabling applications to be "composed" by combining a suitable set of microservices glued together by a lightweight front end.

Technology heterogeneity

Microservices enables development teams to pick the best software technology and language for the implementation of any given application component, without worrying about the rest of the system. Components are loosely-coupled, typically via HTTP or messaging APIs, and this hides their implementation details.

Efficient scaling

Each microservice can be designed to scale out independently of other microservices associated with a given application, which typically means we get more efficient use of resources than with monolithic applications where all functions have to scale in lockstep.

Ease of development and deployment

It's possible to make incremental enhancements to individual microservices and deploy these to production independently of other microservices. If any problems arise from the new version of a given microservices component, the change can quickly be rolled back. This allows for a DevOps approach to the progressive enhancement of an overall application, enabling innovations to be introduced much more rapidly than with monolithic applications which inevitably accumulate many changes between releases, requiring far more comprehensive testing.

Those with a long history in software may be tempted to dismiss microservices as just a new label for Service-Oriented Architecture (SOA), which has been around for many years. There are, of course, many similarities and some practitioners talk about microservices as "fine-grained SOA". The main difference from SOA is the size and scope of the service components: making them fine-grained improves composability, reusability and ease of deployment. Making them too fine-grained may introduce unacceptable inefficiency in the application, so getting the balance right is important.

The microservices approach is not a panacea. Highly distributed loosely-coupled systems bring their own complications, and the complexity of a large application does not disappear just because it is reduced to a set of relatively simple components. Nevertheless, most of the Web-scale players are strongly bought into microservices, none more so than Netflix. Following a disastrous outage in 2008, Netflix started transitioning away from a single monolithic Web application, and has now deployed in excess of 500 microservices to support their Web presence and business operations. Netflix has blogged extensively about its microservices journey, and this material is essential reading for anyone wanting to get under the skin of this approach to system design.

Containers

In the discussion above on the history of virtualization, we described the hypervisor and its support for the deployment of application software in virtual machines. But there is an alternative and more recent approach to virtualization that happens to be particularly well-suited to cloud native applications: Linux containers. In fact containers are now considered indispensable for cloud native applications.

Containers leverage a long-standing method for partitioning in Linux known as "namespaces", which provides separation of different processes, filesystems and network stacks. A container is a secure partition based on namespaces in which one or more Linux processes run, supported by the Linux kernel installed on the host system.

The main difference between a container and a virtual machine is that a virtual machine needs a complete operating system installed in it to support the application, whereas a container only needs to package up the application software, with the optional addition of any application-specific OS dependencies, and leverages the operating system kernel running on the host. Containers rely only on the Linux kernel API which is extremely stable, and identical across different distributions of Linux. This helps to make containers very portable. Containers offer a number of advantages over virtual machines, including the following:

Lower overhead

Because they do not (in most cases) contain complete operating system images, containers have a far smaller memory footprint than virtual machines, and therefore consume considerably less hardware resources. Their small footprint may make it feasible to deploy instances of software to serve single tenants for some kinds of services, and this could simplify the design of the software very considerably.

Startup speed

Virtual machine images are large because they include a complete guest operating system, and the time taken to start a new VM is largely dictated by the time taken to copy its image to the host on which it is to run, which may take many seconds – or even minutes. By contrast, container images tend to be very small, and they can often start up in less than 50 ms. This enables cloud native applications to scale and heal extremely quickly, and also allows for new approaches to system design in which containers are spawned to process individual transactions, and are disposed of as soon as the transaction is complete – an approach which has come to be known as “serverless”.

Reduced maintenance

Virtual machines contain guest operating systems, and these must be maintained, for example to apply security patches to protect against recently discovered vulnerabilities. Containers require no equivalent maintenance.

Ease of deployment

Containers provide a high degree of portability across operating environments, making it easy to move a containerized application from development through testing into production without having to make any changes along the way. Furthermore, containers allow workloads to be moved easily between private and public cloud environments. Being much more straightforward to deploy in the cloud than virtual machines, they are also much easier to orchestrate.

Portability

Applications packaged as containers are highly portable, both across development, testing and production environments, and between different private and public cloud environments. This massively simplifies and speeds up on-boarding of applications compared with VM-based software. It also makes it easy to put in place Continuous Integration / Continuous Deployment (CI/CD) pipelines for acceleration of innovation, and to leverage public cloud services for testing, prototyping, capacity bursting and disaster recovery, offering significant savings in Capex and encouraging experimentation.

Design for Automation

Cloud native applications tend to comprise a substantial number of different software components, partly because they usually implement stateless processing (and therefore have separate components for transaction processing and state storage), and partly because they are usually decomposed into a number of microservices. Furthermore, each microservice is designed to scale out, and so multiple instances of each microservice component need to be deployed to handle the load on the application. For these reasons, deploying a cloud native application at scale may require the instantiation of many tens or hundreds of containers.

It is totally infeasible to carry out the deployment of such an application manually, so cloud native applications are invariably orchestrated in some way so as to automate the deployment process. Likewise, orchestration is needed to automate operations such as scaling of the different microservices and healing failed instances because these would be too complex and onerous to perform manually.

With this in mind, the cloud native application designer pays close attention to the requirements of orchestration and operations automation right from the outset. The main focus is on achieving the simplest possible process for bringing up the components of the application, mainly by minimizing the amount of configuration that needs to be injected into each component. The following practices are commonly employed in cloud native applications to keep things simple from an orchestration standpoint.

Automated IP address assignment

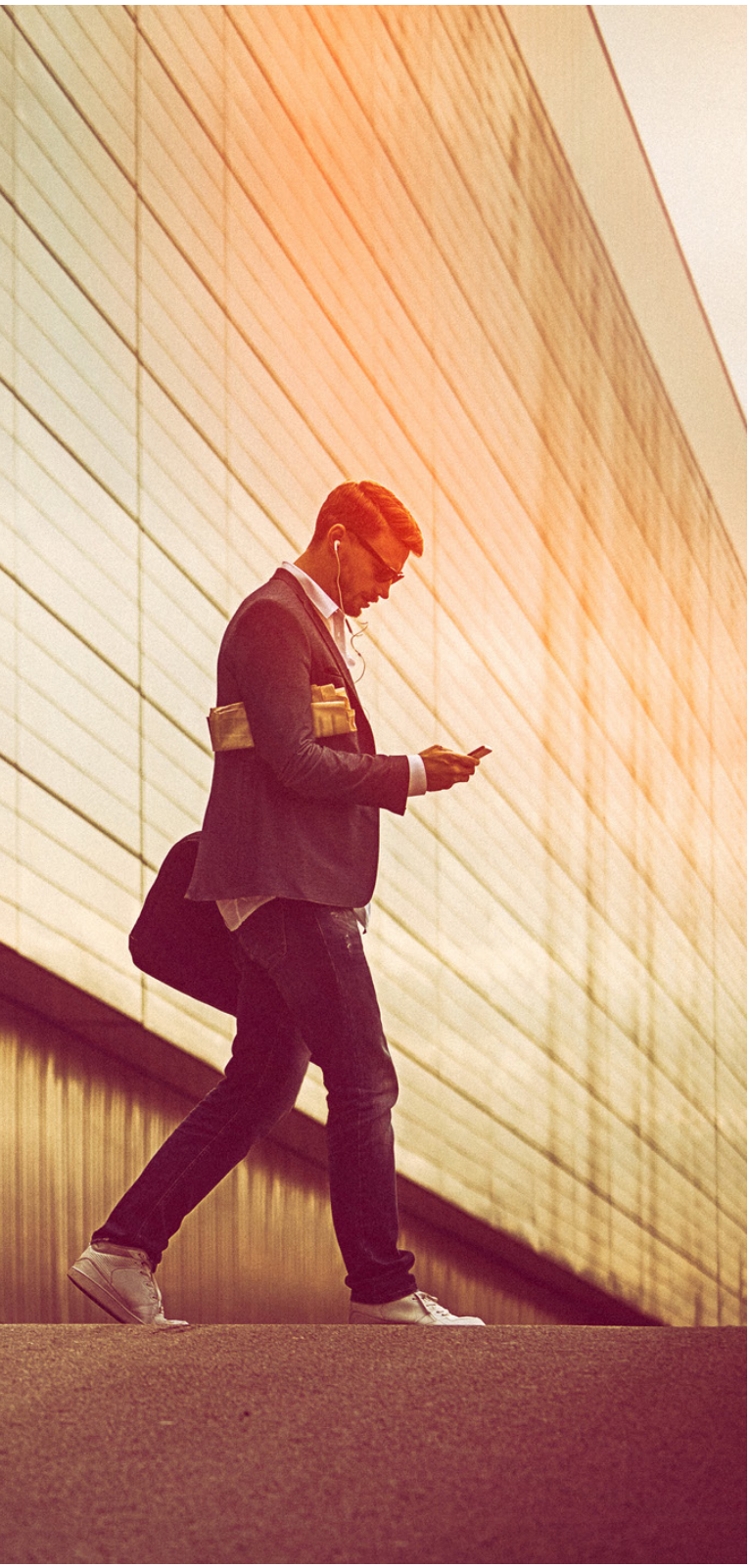
Cloud native application components invariably use DHCP to obtain IP addresses, so the orchestrator does not need to be involved in IP address management. Note that IP addresses can also be hard-assigned to CNF instances where this is necessary.

Shared configuration stores

Cloud native application components invariably participate in a shared distributed key-value store from which they can obtain most or all of the day-zero configuration they need without the orchestrator having to take responsibility for this.

Automated discovery

Cloud native application components typically discover the peers with which they need to communicate either via a shared configuration store or via DNS. Service meshes provide a more advanced means of service discovery.



Elimination of hard dependencies

Many inter-component dependencies typically exist within a given cloud native application, but the components are designed to be brought up in any order. If one component depends on a microservice exposed by another component, and that microservice is not yet available, then the component will keep trying to connect to it until it becomes available.

In the early days of cloud native, a number of different solutions for orchestration, automation and lifecycle management were available, but Kubernetes has emerged as far and away the most popular of these. Kubernetes supports deployment, monitoring, healing, scaling and software upgrade of containerized cloud native applications. Helm provides a means to template the deployment of complex Kubernetes applications in a declarative manner (similar to Heat in OpenStack), while Kubernetes Operators provides a framework for extending the native lifecycle management logic of Kubernetes to custom operations that may be required for more complex cloud native microservices, particularly stateful ones.

It is hard to overestimate the importance of Kubernetes to the cloud native movement. It shows up in every private and public cloud environment on which cloud native applications may be expected to run, and is so universal now that, if an application isn't designed to be orchestrated by Kubernetes, it can't really be considered cloud native.

Building Cloud Native Network Functions

We've discussed cloud native software architecture in the context of Web-scale applications such as messaging, social media and e-commerce, all of which are essentially transactional in nature.

At this point, it is reasonable to ask the question: can these techniques really be applied to the implementation of network functions, given that these are somewhat different in nature to Web-scale applications?

In considering how cloud native principles may be applied to the development of NFs, we need to make a clear distinction between control plane functions and data plane (or user plane) functions.

Cloud Native Control Plane Functions

Control plane functions involve the exchange and processing of messages. For example, routers exchange Border Gateway Protocol messages to learn about the reachability of IP address blocks, and subscribers exchange Session Initiation Protocol messages with an IP Multimedia Subsystem in order to negotiate the establishment of a voice or video session. These functions are transactional in exactly the same sense as the Web-scale applications that we've used as examples of cloud native architecture in action, and all of the cloud native principles can be fully applied to their implementation. Metaswitch's cloud native IMS core solution, Clearwater, is a good example of this.

Cloud Native Data Plane Functions

Data plane functions involve processing packets or packet flows at various levels of the protocol stack. For example, routers forward packets at the IP layer, and may also manipulate packets by terminating tunnels, inserting VLAN tags and so on, while session border controllers forward media packets at the application layer, and may perform various media processing functions such as transcoding. It could possibly be argued that a data plane function is transactional in the sense that each incoming packet represents a "transaction". However, the work done on each packet in a data plane function is typically many orders of magnitude less than the work done in processing a control plane transaction, and simple economics requires us to process many orders of magnitude more packets with a given amount of compute resource compared with control plane transactions. It is impractical to implement a stateless processing model for data plane functions because there is far too much overhead involved in fetching the required state to process each packet from a separate store. The state that we require to process each packet must be locally resident in the network function, in memory or (preferably) in the processor's cache, in order for us to process packets cost-effectively.

State management

In the section above on stateless processing, we explained that the stateless approach enables us easily to scale out an application, and implement fault tolerance with an active-active N+k redundancy model. So if we can't apply stateless processing to data plane functions, does that mean that we can't build a scale-out, active-active N+k data plane function? The answer to this is an emphatic no. By applying appropriate ingenuity in the way we manage and store session and flow state, and how we steer packet flows, we absolutely can build data plane functions that scale out with active-active N+k redundancy. For example, we can divide the state information in any one data plane instance into logical blocks or "shards", and re-distribute these shards across the remaining population of data plane instances when one fails, while modifying the steering of flows to match, for example by leveraging routing protocols or virtual MAC addresses.

Data plane microservices and composability

The next topic we need to consider is whether it makes sense to decompose data plane functions into microservices. We can certainly imagine defining any given data plane function as a sequence of basic actions to be applied to each packet (a packet processing graph), but does it make sense to implement the function with a separate software component for each basic action? The answer to this question depends on exactly how these components are combined together to deliver the complete function. Implementing each basic action as a separately deployable software element in a container, and stringing them together by means of Service Function Chaining or some similar technique, may provide a great deal of flexibility and composability, but it does so at the expense of enormous inefficiency. This is because the work done in the underlying fabric to encapsulate and forward packets between each node of the packet processing graph is likely to be considerably greater than the work done by the packet processing functions themselves. On the other hand, if the software elements that implement each of the basic actions can be composed into a packet processing graph in the context of a single engine, in which packets are passed between components in memory, then we have a "microservices" data plane solution that combines composability nicely with efficiency.

This concept of a composable packet processing engine in which multiple software components that perform basic actions on packets are combined into a single deployable element is gaining currency in the industry. A good example of this is FD.io, an open source packet processing engine, hosted by the Linux Foundation. FD.io enables complex packet processing pipelines to be composed by stringing together code modules, each of which handles some distinct aspect of packet header processing.

A unique advantage of CNAP is its ability to combine multiple logical packet header processing operations into a single composite lookup

Another good example is Metaswitch's Composable Network Application Processor, a proprietary software packet processing engine. This takes a very different approach to FD.io. Instead of building the packet processing pipeline by compiling together a bunch of different code modules, CNAP is entirely configuration-driven. You define the packet processing pipeline you want, in terms of a series of match-action classifier tables, in a YAML document and CNAP uses this information to compose the pipeline on the fly. A unique advantage of CNAP is its ability to combine multiple logical

packet header processing operations into a single composite lookup, enabling it to comprehensively out-perform all other software data plane solutions in the market when applied to complex pipelines such as the 5G User Plane Function.

Compatibility with containers and Kubernetes

High performance data plane NFs leverage a variety of advanced techniques to achieve competitive levels of performance and efficiency, including CPU pinning, hugepage support and SR-IOV. Linux containers have always been compatible with these techniques, but until quite recently Kubernetes did not support them. Recent open source add-ons to Kubernetes, such as the Multus project contributed by Intel, have addressed this issue. As a result, data plane CNFs can now be built that deliver exactly the same level of performance as software running on bare metal servers, while enjoying all of the orchestration and operations automation benefits provided by Kubernetes.

Declarative Configuration of NFs

Most real-world network functions offer hundreds or thousands of configuration parameters that control the detailed aspects of how the network function operates. With traditional network function software, configuration is typically manipulated via a Command Line Interface. Configuration is defined procedurally, by following a sequence of steps towards a desired configuration state, either via the CLI or via a sequence of API calls.

By contrast, the cloud native approach to configuration is declarative. The desired configuration is described in full in a structured document and made available to the CNF, which checks that the requested configuration is internally consistent before applying it. The configuration of a CNF is maintained in a version-controlled repository (typically Git) so that all changes are tracked, and so that the configuration of a CNF can be rolled back to a known good version at any time.

Furthermore, the configuration of VNFs is generally managed independently on each individual VNF instance, whereas with CNFs configuration can be managed globally, or on large subsets of a CNF deployment. By checking in a change to a version-controlled configuration document, the new config is automatically applied to every relevant component instance of the CNF.

The declarative approach to configuration provides far greater control over changes, reduces the likelihood of bad configuration being injected into the network, and greatly speeds up recovery when and if this ever happens.

There has been some movement towards a declarative approach to configuration for some kinds of virtualized network functions with YANG. Nevertheless, procedural approaches to configuration management still dominate the VNF world.

Deployment Environment for Cloud Native Network Functions

In an ideal world, all the NFs we want to deploy would be truly cloud native, packaged in containers and orchestrated by Kubernetes. The ideal cloud infrastructure in this situation would be a native container environment such as Red Hat OpenShift or VMware PKS, running directly on bare metal servers.

The reality, however, is that any cloud infrastructure being built to support the deployment of NFs needs to be able to support both legacy VNFs, packaged in virtual machines, and CNFs packaged in containers – at least for the foreseeable future. Currently, the accepted way of achieving this is to deploy a Kubernetes container environment, such as Red Hat OpenShift or VMware PKS, on top of a hypervisor-based virtualization environment such as OpenStack or VMware vSphere. To do this, you use the OpenStack or vSphere layer to create a pool of virtual machines, and then deploy your Kubernetes cluster supporting your CNFs into that pool.

While this layering may sound inefficient, it works perfectly well in practice. The only downside is that it involves wrestling with two different layers of application orchestration. The CNFs will be orchestrated by Kubernetes, while each VNF will have its own lifecycle manager, typically a Specific VNFM as defined by ETSI.

An intriguing new technology called Kubevirt may provide an elegant solution to this problem in the future. Kubevirt enables any VM-based application to be deployed inside a container and orchestrated by Kubernetes. Once the majority of our NFs are cloud native, Kubevirt may enable us to simplify our cloud environments by eliminating the hypervisor layer while still being able to deploy and manage legacy VM-based VNFs. All orchestration is then performed by Kubernetes

Testing Cloud Native Claims

By comparison to the traditional approach to NFV based on VNFs, deploying and operationalizing CNFs is vastly easier and provides a far clearer path to realizing the promised benefits of NFV including substantial Opex and Capex reductions, and rapid acceleration of innovation. Consequently, there is a strong temptation for vendors to re-position their VNF products as CNFs, without much regard for observing the cloud native principles we have described. Network operators therefore need to be very wary of claims being made by vendors that their NF products are cloud native.

No Evolution Path from VNF to CNF

In general, it is extremely difficult to refactor monolithic, stateful, legacy network function software so as to embody the key architectural aspects of cloud native that are essential to the delivery of cloud native benefits. The main reasons for this are as follows:

State is inextricably tied into the code

In traditional application architectures, all state is stored locally and elements of state are accessed or updated by individual instructions throughout the body of the code. Unpicking this so as to read all relevant long-lived state from a separate store and to update that store at the relevant points in the processing of any given event is hugely labor-intensive, and in most cases represents an almost complete re-write of the code.

Monolithic applications are hard to de-compose

While legacy codebases typically show a high degree of modularity in the form of function calls and subroutines, these modules rarely offer natural boundaries for decomposition to loosely-coupled microservices. Often this is because of mutual dependencies on shared data structures. A microservices architecture requires such dependencies to be eliminated, and this usually demands a complete re-think of the application architecture.

Procedural configuration management is very different to declarative

The key issue here is that, with procedural configuration management, consistency checking is applied in a stepwise manner, with each step dependent on what has gone before. By contrast, declarative configuration requires holistic consistency checking of the entire configuration before any change is applied. These two approaches are fundamentally different, and it requires a great deal of effort to change a complex application from one to the other.

Any claim being made by a vendor that a network function is cloud native when it is quite obviously derived from a legacy NF codebase should therefore be treated with great caution.

The Cloud Native Scorecard

Network operators should arm themselves with a series of questions that are designed to test the veracity of any cloud native claims being made for a software NF product. A genuinely cloud native NF should be able to answer the great majority of these questions in the positive.

Is the software packaged as containers and orchestrated by Kubernetes?

While historically it has been possible to embody cloud native architectural characteristics in VM-based applications and orchestrate with a Generic VNF Manager, containers and Kubernetes have come to be synonymous with cloud native and should be regarded now as absolute requirements for a CNF.

Is the software composed from multiple microservice components, each of which exposes an open, versioned and documented language-agnostic API?

Vendors should be prepared to share details of their microservices as re-usable building blocks, and explain in detail what function each microservice provides.

Are the microservices truly loosely-coupled?

If two or more microservices access state in a shared store where each microservice relies on a common schema for the shared state, then these microservices must be regarded as tightly-coupled. This is a cloud native anti-pattern since it introduces dependencies that prevent microservices from being independently enhanced.

Does the NF comprise a mix of stateless microservices (for event or message processing) and stateful microservices (for storage of long-lived state)?

All CNFs that relate to control plane functions should exhibit this design pattern, which is essential for straightforward automation of scaling, healing and software upgrade. Data plane CNFs will necessarily be stateful.

Do the individual microservices scale out dynamically by adding software instances under the automatic control of Kubernetes?

All stateless microservices should scale out in a very simple fashion by adding new software instances, without requiring complex initialization or configuration steps to bring up each new instance. Note that scaling of stateful microservices is typically more complex, but this should still be handled automatically by Kubernetes with the aid of custom Kubernetes Operators.

Are the individual microservices fault tolerant according to an active-active N+k redundancy scheme?

A CNF should be resilient to the loss of an individual software instance, simply by providing a small amount of surplus capacity. It should never require 1+1 active-standby protection. The only exception to this rule is for microservices that terminate legacy network protocols where the endpoint is defined by a fixed IP address. In those cases, 1+1 protection with virtual IP address swapping is the only way to achieve high availability.

Can a failed software instance be recovered simply by killing it and instantiating a new one?

The health of individual software instances in a CNF is monitored by Kubernetes, which expects to be able to kill and replace the instance if it misbehaves. Complex healing procedures should never be required.

Can software upgrades be applied to individual microservices in a non-service affecting manner by a process of rolling update?

If the microservices are loosely-coupled and the APIs they expose are properly versioned, then automated in-service software upgrades can be applied easily and automatically under the control of Kubernetes.

Is NF configuration managed declaratively?

A CNF should not expose a CLI, nor should it rely on procedural configuration management via sequences of API calls. Instead, configuration should be declared in documents maintained in version-controlled repositories, where checked-in changes are applied automatically to all the software instances that make up the CNF.

Does the NF expose a comprehensive set of instrumentation APIs that are compatible with the appropriate cloud native tools?

Prometheus and Fluentd are now so ubiquitous in the cloud native world for collection of metrics and logs that all CNFs should implement native APIs towards these tools. CNFs should also expose tracing information that provides visibility of all message and event processing activities, but given the lack of suitable open source solutions for most NF use cases, proprietary tracing collection and analysis solutions are acceptable.

Conclusion

Cloud Native Network Functions (CNFs) are very different from Virtualized Network Functions (VNFs) in their design, architecture and relationship to the open source software ecosystem. It is these differences that enable CNFs to deliver on the full promise of network virtualization, unlike VNFs: substantive Opex efficiencies, reduction in service-affecting faults, lower Capex through improved hardware utilization, and acceleration of innovation through adoption of DevOps approaches.

But the technical differences between CNFs and VNFs run so deep that it simply isn't possible to envisage a feasible evolution path for a VNF to become a CNF. To build a CNF, you really need to start from scratch. That requires a huge investment, and it isn't very surprising that many vendors are taking the easy way out, and simply attaching a CNF label to products that are really VNFs.

Armed with the knowledge provided by this paper, network operators will be in a much better position to assess to what extent any given NF product is really cloud native, and therefore what likelihood it has of bringing the truly transformative improvement that the cloud native approach is capable of delivering.